TP02 - Estrutura de dados

Marcos Daniel Souza Netto - 2022069492

November 13, 2023

1 Introdução

O problema proposto consiste em implementar um programa que recebe um grafo e a coloração dos seus vértices e verifica se a coloração dada é gulosa, ou seja, todo vertíce deve possuir a cor i somente se existir vertices adjacentes a ele com cores $1, 2, ..., i-1$. Por fim, também deve ser retornada a permutação dos vértices utilizada na coloração gulosa, isto é, apresentar como o grafo foi colorido, apresentando os vértices de cor i seguidos dos vértices de cor $i+1$, os vértices de cada cor devem estar ordenados pelo seu rótulo.

A solução implementada leva em consideração a quantidade de vértices mínima de diferentes cores, menor que a cor do vértice analisado, para que a coloração seja gulosa. Observe que para satisfazer condição de gulosidade é necessário que cada vértice de cor i tenha pelo menos $i-1$ vértices adjacentes de cores menores que i. Para isso, foi utilizada a estrutura de dados **set**, tal que para todo vértice de cor menor que i, é inserido sua cor no set, e depois de passar por todos essas cores, é verificado se o tamanho do set é menor que $i-1$, caso seja, a coloração não é gulosa.

Para a parte da permutação usada, apenas foi utilizada a ordenação escolhida de modo que a comparação entre dois vértices é feita pelo sua cor primeiramente, e caso sejam iguais, é feita a comparação pelo seu rótulo.

2 Método

A seguir serão detalhados a implementação da solução, de modo a explicitar as estruturas de dados utilizadas e as estratégias de solução. A priori vale apresentar as especificações do ambiente de desenvolvimento utilizado:

- Sistema Operacional: Ubuntu 22.04.3 LTS;
- Compilador: G++ 11.4.0;
- Processador: Ryzen 5 5500u;
- Memória RAM: 8GB.

2.1 Estruturas de dados

Em síntese, foram utilizadas as seguintes estruturas de dados:

- **set**: Estrutura de dados que armazena elementos únicos, e que permite a inserção, remoção e busca em tempo $O(log(n))$, para tanto ela foi implementada como uma árvore binária de busca, e foi utilizada para armazenar as cores dos vértices adjacentes;
- **vector**: Estrutura de dados que armazena elementos em sequência contínua de memória. Foi utilizada para representar o grafo, ou seja, o grafo aqui foi representado como um vetor de listas de adjacências, onde cada posição do vetor representa um vértice, tal que cada posição possui um subvetor contendo os vértices adjacentes a ele;
- **pair**: Tipo abstrato de dados que generaliza a ideia de um par ordenado. Foi utilizada para representar a coloração do grafo, o primeiro elemento do par é a coloração do vértice, e o segundo elemento é seu rótulo. Permitiu que a construção da permutação usada na solução seja feita de forma mais simples, pois a ordenação no *vector* de *pairs* é feita pelo primeiro elemento do par, e caso sejam iguais, é feita a ordenação pelo segundo elemento;

A implementação dessas estruturas se localizam nos arquivos *set.hpp*, *vector.hpp*, *graph.hpp* e *pair.hpp*.

2.2 Classes e principais funções

O código foi dividido nos arquivos já mencionados (*set.hpp*, *vector.hpp*, *graph.hpp* e *pair.hpp*), e também no arquivo *sort.hpp* dos métodos de ordenação implementados e o arquivo principal *main.cpp*.

- **set.hpp**: Implementação da estrutura de dados *set*, por meio de uma árvore binária de busca não balanceada, que foi utilizada para armazenar as cores dos vértices adjacentes. Foram implementados os principais métodos da estrutura, como: *insert*, *remove*, *find*, *getSize*, *walk* e *clear*;

  walk: Percorre o *set*, e para cada elemento, chama uma função passada como parâmetro;

- **vector.hpp**: Implementação da estrutura de dados *vector*, um array de memória contíguo como tamanho podendo ser aumentado e controlado pela própria estrutura. Foram implementadas as principais funções da estrutura, como: *push_back*, *pop_back*, *insert*, *getSize*;

- **sort.hpp**: Implementação dos métodos de ordenação utilizados.

  bubleSort: Consiste em percorrer o vetor, e para cada elemento, percorrer o vetor novamente, e caso o elemento atual seja maior que o próximo, eles são trocados de posição;

  insertionSort: Consiste em percorrer o vetor, e para cada elemento inseri-lo na posição correta do subvetor ordenado;

  selectionSort: Consiste em percorrer o vetor, e para cada elemento, percorrer o subvetor não ordenado, e encontrar o menor elemento, e trocá-lo de posição com o elemento atual;

  mergeSort: Consiste em dividir o vetor em dois subvetores, ordenar cada um deles e depois juntá-los de forma ordenada;

  quickSort: Consiste em escolher um pivô, e colocar todos os elementos menores que ele a sua esquerda, e todos os elementos maiores que ele a sua direita, e depois ordenar recursivamente os subvetores a esquerda e a direita da partição criada;

  heapSort: Consiste em construir uma *heap* a partir do vetor, e depois retirar o elemento do topo da *heap* e colocá-lo no final do vetor, e repetir esse processo até que a heap esteja vazia;

  swap: Função que troca dois elementos de posição no vetor;

- **pair.hpp**: Tipo abstrato de dados de par ordenado, com os elementos *first* e *second*. Foi implementadas as funções de comparação específicas para o problema proposto (Comparação pelo primeiro elemento, e caso sejam iguais, comparação pelo segundo elemento);

- **graph.hpp**: Tipo abstrato de dados que representa um grafo por uma lista de adjacência utilizando a estrutura *vector*.

- **main.cpp**: Arquivo principal do programa, onde é feita a leitura da entrada, e a chamada das funções que resolvem o problema proposto.

  sort: Função que recebe um vetor de *pairs* e um método de ordenação, e ordena o vetor utilizando o método de ordenação passado como parâmetro;

  verifyGreedy: Função que recebe um grafo e uma coloração, e verifica se a coloração é gulosa, retornando *true* caso seja, e *false* caso contrário;

3 Análise de Complexidade

A seguir serão apresentadas as análises de complexidade das principais funções do programa utilizadas na solução.

3.1 infixToPosfix

Como mencionado acima, esta função converte uma expressão infixa para a representação posfixa. Para isso, a função percorre a string contendo a expressão infixa, e para cada caractere, verifica se ele é um operando ou um operador. Caso seja um operando, ele é adicionado à string de saída, caso seja um operador, ele é adicionado à pilha. Ao final da leitura da string, todos os operadores que estão na pilha

são adicionados à string de saída. Portanto, a complexidade da função é $O(n)$, onde n é o tamanho da string de entrada.

A complexidade de espaço é $O(n)$, pois a pilha pode conter no máximo n elementos.

3.2 evaluateExpression

Esta função recebe um vetor de inteiros representando a expressão booleana na forma posfixa e um vetor contendo os valores das variáveis da expressão e retorna o resultado da expressão. Para isso, a função percorre esse vetor de entrada, e para cada elemento, verifica se ele é um operando ou um operador (inteiros negativos). Caso seja um operando, ele é adicionado à pilha, caso seja um operador, ele é removido dois operandos do topo da pilha ou apenas um operando (No caso do operador $\sim$), e o resultado da operação é adicionado à pilha. Ao final da leitura da string, o resultado da expressão é o elemento que está no topo da pilha. Portanto, a complexidade da função é $O(n)$, onde n é o número de elementos do vetor de entrada.

A complexidade de espaço é $O(n)$, pois assim como a função anterior, a pilha pode conter no máximo n elementos.

3.3 satTree

Esta função recebe um vetor de inteiros representando uma expressão booleana na forma posfixa, um vetor contendo os valores das variáveis da expressão(agora podendo conter quantificadores existe e para todo) e retorna 1 ou 0 caso a expressão seja satisfazível ou não, respectivamente. Caso ela seja satisfazível, uma string contendo a possível solução é retornada.

Para isso, a função indexa os quantificadores da expressão, em seguida percorre o vetor dos quantificadores construindo a árvore de solução pelos últimos quantificadores até o primeiro. A árvore possui, portanto, como folhas os possíveis valores para os quantificadores (0 ou 1) e desse modo ela possui 2^k folhas, onde k é o número de quantificadores, seja existe ou para todo. Para exemplificar considere a figura.

Como visto, a árvore possui 2^k folhas, e para cada uma delas, a função *evaluatePostfix* é chamada para verificar se a expressão é satisfazível. Portanto, a complexidade da função é $O(2^k)$, onde k é o número de quantificadores.

Já a complexidade de espaço é $O(k)$, pois a solução da árvore é realizada de modo ir o mais profundo na árvore possível, e cada vez que um nó é resolvido, seus filhos são removidos da memória.

4 Estratégias de robustez

A estratégia de robustez utilizada foi basicamente a verificação da entrada do usuário, ou seja a verificação na quantidade de argumentos e seus tipos.

5 Análise Experimental

A seguir serão apresentados os resultados obtidos na experimentação do programa.

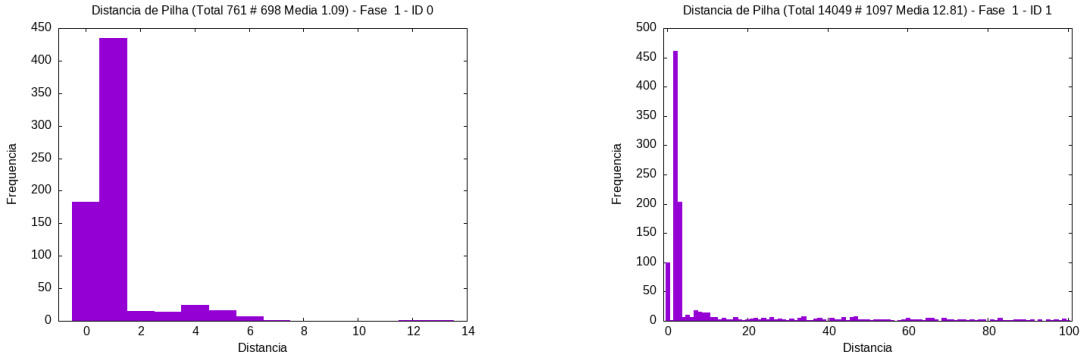
5.1 Localidade de Referência

Utilizando o programa analisamem, fornecido pelo professor, foi possível observar a localidade de referência do programa. Importante notar que aqui serão apresentadas imagens relacionadas a parte de satisfatibilidade do programa, afinal utiliza a parte de solucionador de equaações booleanas como subrotina.

Ademais, foi utilizada a seguinte expressão para a parte de satisfatibilidade:

```
./bin/tp01 -s "0 | 1 & 2 & 3" e11e
```

5.1.1 Verificação de Gulosidade



(a) Lista de adjacência do grafo

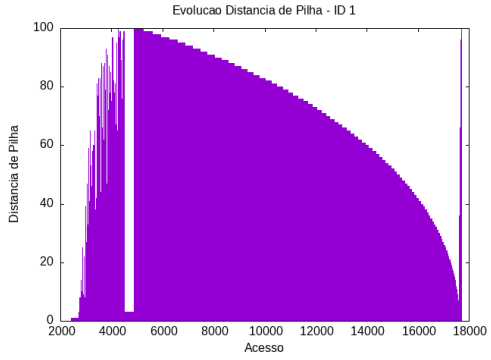
(b) Vetor de cores

Figure 1: Frequência de distância de pilha na verificação de gulosidade

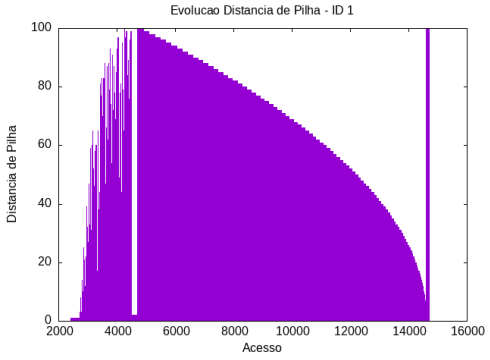
Observe que o algoritmo de verificação de gulosidade é bem simples, e ainda possue uma boa localidade de referência, uma vez que em geral estamos trabalho com espaços de memória contíguos, e esses acessos são feitos de forma sequencial. Distâncias de pilha maiores significam que estamos no geral mudando o vetor de adjacência que estamos percorrendo.

5.1.2 Ordenação

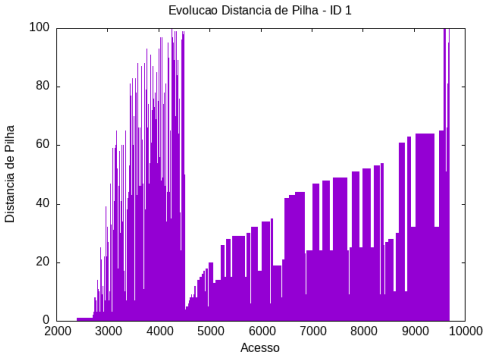
Aqui, vamos detalhar principalmente sobre a localidade de referência do método de ordenação próprio, afinal os demais métodos foram implementados nas suas versões clássicas. Fica como sugestão a análise dos demais métodos.



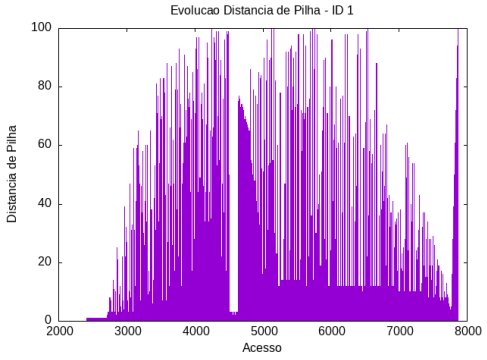
(a) Método por bolha



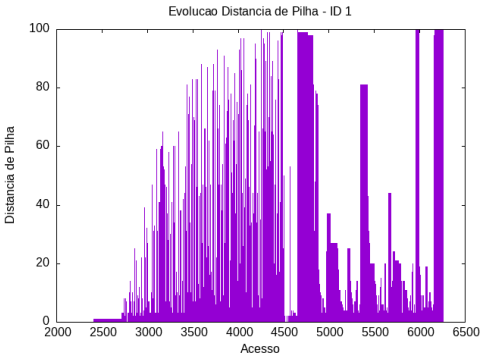
(b) Método por Seleção



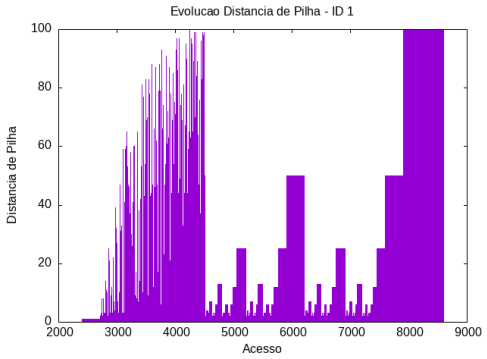
(c) Método por Inserção



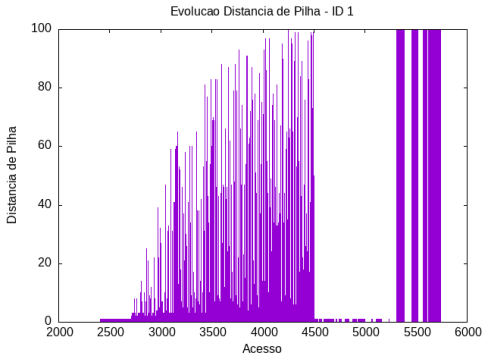
(d) Método do Heap



(e) Método do QuickSort



(f) Método do MergeSort



(g) Método próprio

Figure 2: Frequência de distância de pilha nos métodos de ordenação

Observe que a figura 2g (Ordenação própria) se assemelha bastante à figura 2f (MergeSort), isso se deve porque a ordenação própria possui a primeira etapa de separar os vértices, como já descrito anteriormente, e o mergesort com o processo de partições. E na segunda etapa, ambos possuem a mesma ideia de percorrer os subvetores ordenados e atribuir os elementos para o vetor original, que no caso do

mergesort é repetido diversas vezes. Pode-se se certificar desse comportamento devido às "barras" de mesma distância de pilha presentes no final do gráfico.

6 Conclusões

O trabalho proposto foi de grande valia para o aprendizado de estruturas de dados, e de como elas podem ser utilizadas para solucionar problemas, afinal foi de importância considerável a utilização de árvores (*set*) e a representação como lista de adjacência de um grafo como estruturas principais, para se ter uma complexidade menor no algoritmo como todo. Ademais, temos também o uso dos métodos de ordenação, mas que nesse trabalho ficaram em segundo plano.

Bibliografia

Slides da disciplina de Estrutura de Dados, ministrada pelo Prof. Wagner Meira Jr. e Prof. Eder Fereira Figuiredo.

Instruções para compilação e execução

É estritamente necessário ter instalado na máquina o compilador g++, com pelo mínimo a versão descrita no começo do trabalho.

Em um terminal, navegue até a pasta raiz do projeto e execute os seguintes comandos:

```
$ make
$ ./tp02.out
```

Agora, basta seguir o padrão de entrada descrito na especificação do trabalho.